2009:122 CIV

MASTER'S THESIS

Requirements and possibilities of a new microcontroller in an embedded system

David Wiberg

Luleå University of Technology

MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of EISLAB

2009:122 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--09/122--SE

ABSTRACT

Shortening product development cycles and reducing cost are important aspects for all development work. Being able to reuse resources spent on earlier product generations is one way to lower costs by preventing the same job from being done multiple times. In this master's thesis in Computer Science and Engineering an evaluation of a new microcontroller is made with the goal of reusing a large codebase from previous projects. Studied is also the currently used program loading methods at Haldex Traction and a new concept is suggested which would render some of the current limitations invalid. Finally a presentation of the practical work needed to adapt a small real-time kernel called TinyTimber to a new platform is explained.

PREFACE

This master's thesis in Computer Science and Engineering is written at Luleå University of Technology with Per Lindgren as examiner.

The possibility to write my master's thesis at Haldex Traction was given to me with help of my good friend Gustaf Lagunoff. For this I send my gratitude since the time spent at Haldex has been rewarding in many ways.

I would also like to thank Johan Nilsson and Adam Eliasson at Haldex who have been supervising my work and always found the time to answer questions and provide feedback.

Finally I would like to thank my brother who has helped me by providing a lot of feedback.

TABLE OF CONTENTS

1 Introduction 1
1.1 Objectives 1
1.2 Delimitations 1
2 Current Platform 2
2.1 Overview 2
2.2 Mechanical Principle 2
2.3 Software 4
3 New platform 8
3.1 Introduction 8
3.2 Methodology 8
3.3 Microcontroller 8
3.4 Software 10
3.5 Conclusions and discussion 12
4 Program loading 13
4.1 Overview 13
4.2 Available methods 13
4.3 Current solution 13
4.4 Suggested solution 14
4.5 Conclusions and discussion 15
5 JTAG-Debugging 16
5.1 Introduction 16
5.2 Debug 16
5.3 Components 16
5.4 Alternatives 17
5.5 Evaluation of different JTAG debugging systems 17
5.6 Conclusions and discussion 23
6 TinyTimber 24
6.1 Introduction 24
6.2 Software structure 24
6.3 Time management 24
6.4 Creating a new environment 25
6.5 Details of the XC2000 environment 27
6.6 Discussion 28
References 29

1 Introduction

Haldex Traction is a company which develops and manufactures an All Wheel Drive (AWD) system for passenger cars. The system is controlled by an Electric Control Unit (ECU) and Haldex wants to examine the possibility to replace the microcontroller in the ECU for an upcoming product generation. To support the decision process the author has studied both a microcontroller considered as an option for new product development and the current product and its related development tools. New tools available to support development on the suggested microcontroller have been studied to be able to provide a more complete picture of the effects a change will have.

Related to the development process for an embedded application is the question of how the actual software application is loaded to the ECU. Since the currently used implementation has got some limitations, available options have been studied to find a way to circumvent them.

Finally a real-time kernel called TinyTimber is studied. TinyTimber is an active research subject at Luleå University of Technology (LTU) and the kernel has been used by the author in courses at LTU.

1.1 Objectives

The objectives of this master's thesis regarding the new microcontroller are:

- Make sure that it's suitable for new product development regarding performance and features.
- Examine what's needed to run the currently used real-time operating system.
- Evaluate how much work is needed to port the current application to the new architecture.
- Examine which new tools are needed and to what extent old tools can be reused.
- Examine available options for program loading.
- Create an environment which allows running of a real-time kernel called TinyTimber.

1.2 Delimitations

The new microcontroller studied in this report is chosen by Haldex, limiting the comparison to two different products. The microcontrollers have been studied on a high level focusing on the provided functionality and not on internal structure.

The part of the software studied in the current application is situated closest to the hardware. The higher levels which contain models for vehicle dynamics have not been considered.

Regarding program loading a demonstrator has been created but it works as a proof of concept and is not suitable for production use.

Due to time constraints the work with TinyTimber has been limited to practical porting of the system and a report of the experiences learnt in the process to provide a basis for similar work in the future.

2 Current Platform

2.1 Overview

In this chapter the latest generation of Haldex Traction's All Wheel Drive (AWD) system is studied to provide a basis for further discussion. With an AWD system torque can be sent from the engine to all wheels simultaneously. The benefits of this are most noticeable for the average driver when the road conditions are slippery, e.g. when there is mud, water or snow on the road. There are a number of different design choices available for an AWD system and Haldex has chosen one called "hang on" AWD. This concept means that a front or rear wheel drive vehicle is able to transfer torque to the normally non driven axle when the situation calls for it.

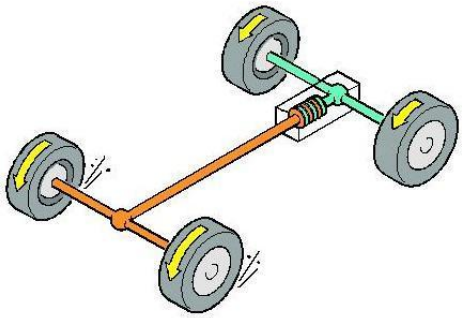

Figure 1 - The front axle is connected to the rear axle

Using a Haldex Limited Slip Coupling (HLSC) is one method which can be used to connect the two axles and distribute the engine torque between them. The HLSC could be described as a wet multi-plate clutch which is operated by hydraulic pressure and controlled by an electronic unit. Four different generations of the coupling have been created and the mechanical complexity has been reduced with each new generation. The software controlling the HLSC has been made more advanced to get the same amount of performance from fewer components which reduce costs. The product examined in this section is Haldex LSC Generation IV.

2.2 Mechanical Principle

The coupling is operated by a hydraulic system which is shown in the Figure 2. Oil lines with high pressure are drawn with red color; pink color represents reduced pressure and white shows lines without pressure. The black arrows show the direction of oil flow.

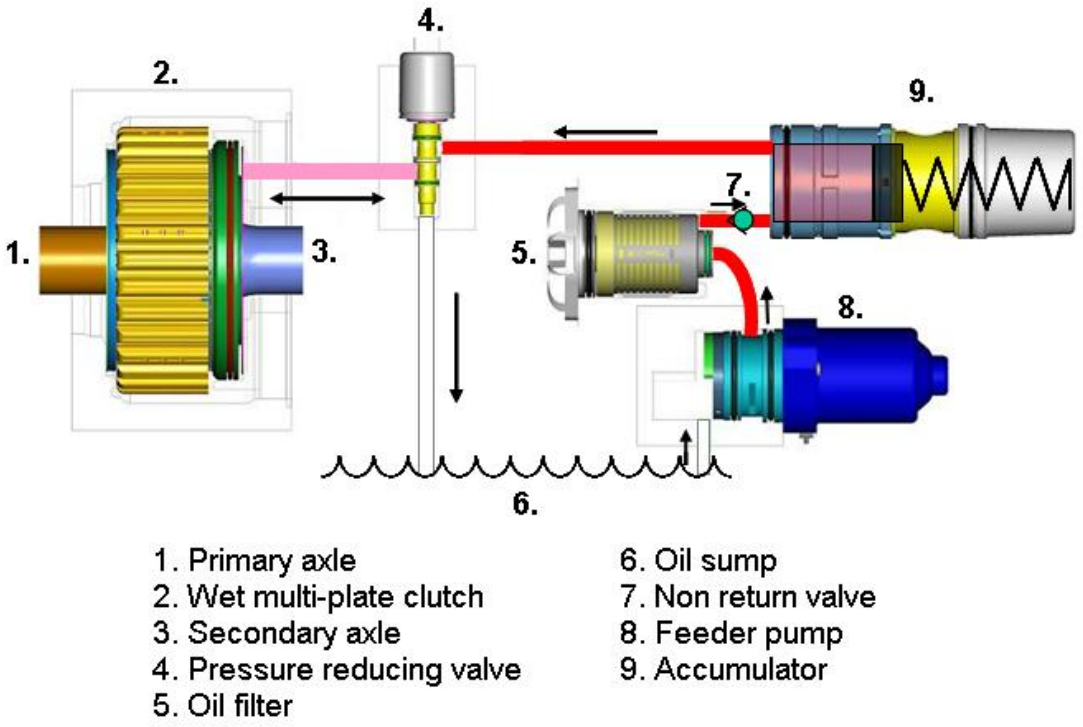

Figure 2 - Mechanics of Haldex LSC

The principle of operation is that the feeder pump pumps oil through the filter to an accumulator. The accumulator contains a piston which is spring loaded to keep a constant pressure when the volume of oil in the accumulator changes. The pressure reducing valve is used to control the pressure which affects the clutch. With increased pressure the amount of torque transferred from the primary axle to the secondary axle is increased.

2.2.1 Electronics

The Electronic Control Unit (ECU) is the unit of the coupling which is responsible for controlling the hydraulic system. It does so by controlling the feeder pump and the pressure reducing valve. The central component of the ECU is a microcontroller unit (MCU) from the company STMicroelectronics, commonly called ST. The name of the MCU is ST10F272M and it's a 16 bit processor with a clock frequency of 40MHz[1].

[1] STMicroelectronics, *Datasheet ST10F272M*, 2008, retrieved May 16, 2009, <http://www.st.com/stonline/products/literature/ds/12968.pdf>

2.3 Software

2.3.1 Overview

The software running in the ECU is structured according to Figure 3. At the very bottom of the figure is the actual hardware, i.e. the electronics. Closest to the hardware is the base software which is responsible for steering of the pump and the release valve. This layer is mainly the same between different customers except for diagnostics and communication drivers which are customer dependant.

The Control Software keeps track of the current driving situation and sends requests to the base software to change the amount of torque transferred from the primary axle to the secondary axle. All changes needed to adapt the behavior of the coupling for a new car model should be contained in the calibration parameters.

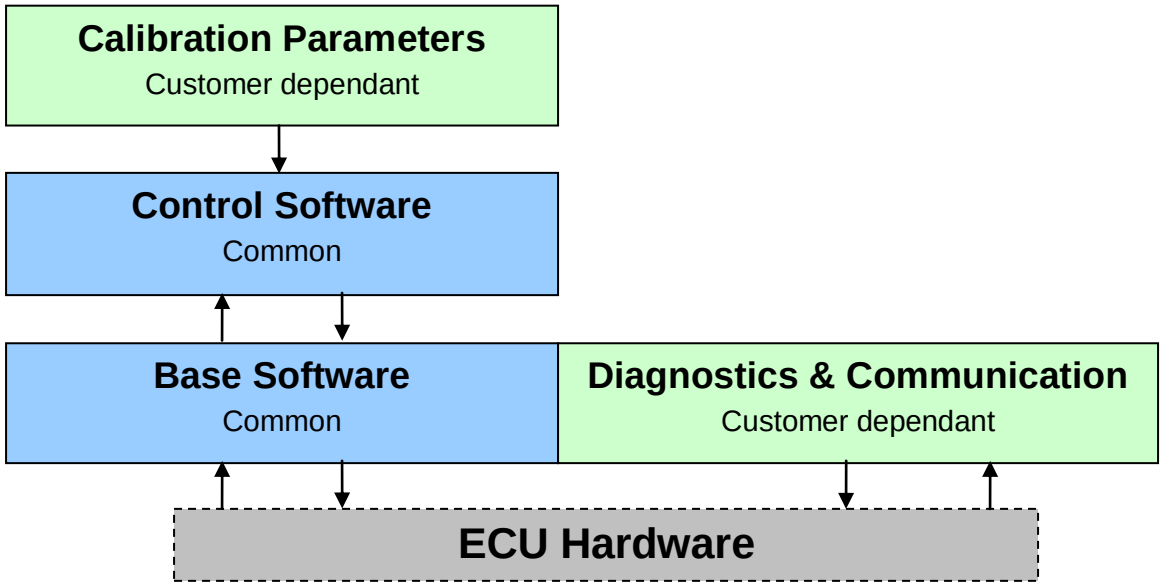

Figure 3 - Software layout

Haldex is currently using a real-time operating system called Rubus OS from the company Arcticus Systems. The following description is based on the author's experience with the OS and information provided by Arcticus Systems (See website *http://www.arcticus-systems.com* for more information).

Rubus OS provides an execution platform for an application in a time critical system. This means that the operating system can execute different parts of an application according to specifications given by the developer. This is made possible by the introduction of two different kinds of services. One for tasks which have defined hard deadlines; these will be statically

scheduled pre-runtime and this is called the *red service*. The other is called *blue service* and is for tasks which are event triggered and will be run according to a priority based scheduling policy.

A typical area of usage for the *red services* is for tasks which are run periodically and must also start and finish execution within some declared time constraints. An example could be a software based stopwatch which must update its value once every second. The event triggered execution model provided by the *blue services* is used in an application to react to some kind of event. Using the same stopwatch as example, a typical event can be that a user pushes a button and the stopwatch can react for example by resetting the time to zero.

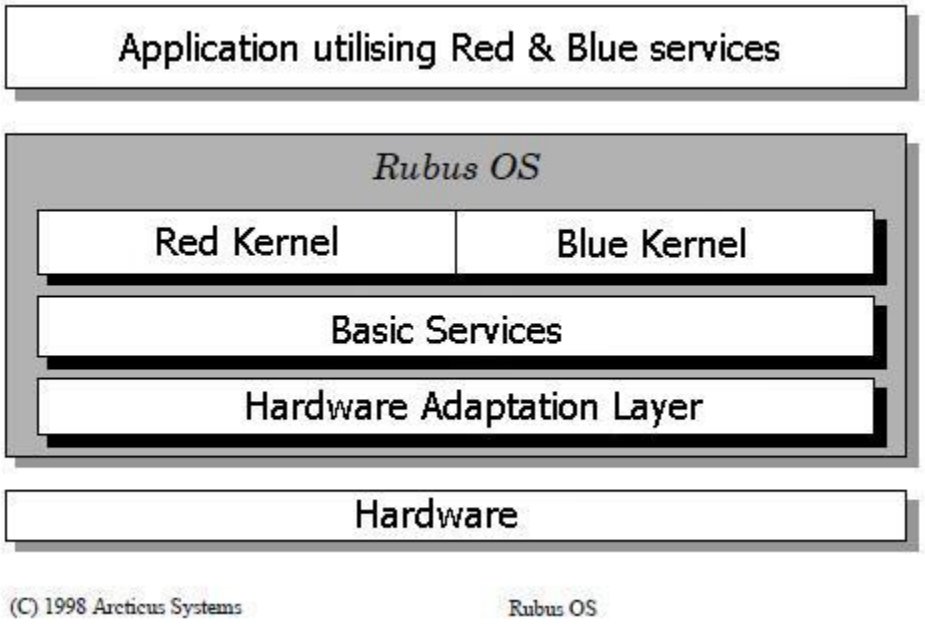

Figure 4 - Rubus OS overview

The overall structure of the Rubus OS is shown in Figure 4. Closest to the user application are the *red*- and *blue kernels* which provide the *red*- and *blue* services described above. The b*asic services* supplies features which are common to both kernels, for example here the treatment of time and interrupt handling is introduced. A Hardware Adaptation Layer (HAL) is situated closest to the actual hardware. It includes all processor dependent code to facilitate porting of the OS to different processor architectures.

The notion of time in Rubus OS is based around the *basic clock* which is used to measure the passage of time. The value of this basic clock is updated by a periodic hardware timer in the microcontroller. The clock is coupled to a timer called the *red timer* and it's used to create schedules for red threads. A schedule in Rubus OS describes when different threads can start execution and when they must be finished. The time from the start of a schedule to the end is the period time. When the active schedule has reached the period time the *red timer* expires

and is reloaded with zero and the schedule restarts. The time constraints for a thread are expressed relative the start of a schedule.

It's possible to create and use several different schedules inside a single application. This can be used to set different application modes depending on the current situation. One common example is to use one schedule which handles startup of an application and switch to another one for normal operation.

Time in the blue kernel is measured by the *blue clock*; its value is periodically updated from the basic clock.

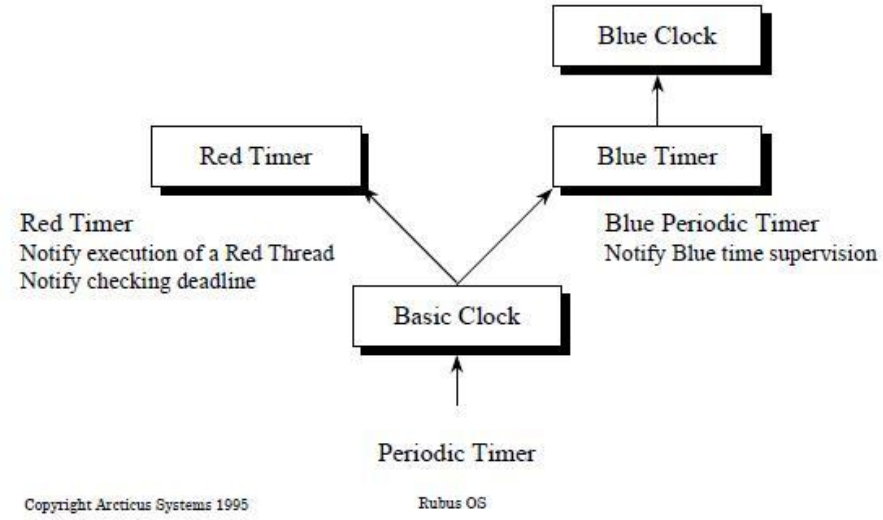

Figure 5 - Time in Rubus OS

2.3.3 Base Software

The base software is considered to consist of three different parts, Hardware Adaptation Layer (HAL), Strategic Control and Diagnostics. These three parts combined are also often called platform or base layer. The description in this section is based on a code review and reading of Haldex internal documentation.

The HAL is the part of the application which operates closest to the hardware and provides a high level API to it. It's for example responsible for current control of the valve, complete control of the base pressure and operation of the oil pump. It also provides communication possibilities over the Controller Area Network (CAN), which is described in ISO standard 11898, and is the most common method to allow different units in a modern car to communicate with each other. The HAL also contains libraries used for mathematical and filtering functionality.

The strategic control continuously monitors the base software and hardware during operation. Status information from the different subsystems is retrieved via function calls and based on the

gathered information the strategic control manager determines the operating mode of the ECU. The different modes are implemented by different schedules in the Rubus OS.

The diagnostics part collects and stores error information from other software components. It's possible to send commands to this part of the software by connecting a test device to the diagnostics bus in the car. The diagnostic application can take over control of the coupling by asking the strategic control to put the ECU in a special mode.

The modules in the base software are classified according to a system where a *driver* is a software component which works as a layer between software and hardware. It's only responsible for passing data to and from the hardware device. A *manager* manages at least one specific task in the system. A d*evice* is a sub-system which consists of one or more components which together handles a hardware device. There are a few more possible classifications but these are the ones closest related to the hardware.

To exemplify these definitions a device called *valve device* has been chosen. Closest to the hardware is the *valve driver* which contains functionality to set the duty-cycle of a pulse width modulated (PWM) signal which controls the valve. The driver is controlled by the *valve control manager*. It provides an interface for setting the pressure of the system and maintains it by means of a software regulator. Finally the *valve device* gives other software components a possibility to set the pressure and is also responsible for error detection of the valve hardware.

To be able to communicate with other units in a car the ECU utilizes the CAN-bus of the vehicle. The ISO standard for CAN does not define how higher level application tasks such as flow control and transmission of data which doesn't fit in a single message should be implemented. Since there are many units from different manufacturers connected to the CAN-bus in a car each car-manufacturer specifies a strict set of communication rules which must be obeyed. To ensure this and avoid certification costs a third-party CAN implementation is used.

3 New platform

3.1 Introduction

Haldex Traction is working with the planning of the next generation of HLSC called Generation 5P. One of the MCUs considered for this new generation comes from Infineon and is called XC2734. Part of the author's task has been to investigate some of the features of this processor and tools available to use them. Focus has also been put on a study of how the base layer of the current application will have to be adapted to be suitable for the new MCU.

3.2 Methodology

A study of the Rubus OS has been performed by reading available documentation and looking at the current application to see how it cooperates with the OS. This has been done to be able to implement a relevant software example for the new MCU. An evaluation board with a MCU from the same series has been used to provide a demonstration platform.

To be able to present an opinion of how much work is needed to port the base layer a study of the software components has been performed. This has both helped determine which parts of the software that contains hardware specific code and also to see which features of the MCU are used by the application.

During the entire work with the demonstrator applications different debugging solutions has been continuously tested to form an experience basis for an assessment of their capabilities.

3.3 Microcontroller

3.3.1 Overview

The new microcontroller considered in this report is the XC2734 from Infineon. It's part of the XC2000-series of 16-bits microcontrollers from Infineon. This series of MCUs all share the same CPU core called C166S V2. The differences between models in the series lie in operating frequency, amount of memory and available peripherals.

3.3.2 Specifications

The C166S V2 core is derived from the successful C166 microcontroller family which was introduced in the beginning of the 90's. The currently used processor from ST belongs to the same family which has grown popular and today nearly one billion devices have been sold. The C166S V2 core is instruction set and code compatible with the C166 family which enables reuse of existing code. [2]

[2] EmbeddedDeveloper.com, *Infineon C166®S V2,* retrieved April 3, 2009, <http://www.embeddeddeveloper.com/cores_variant/66/Infineon-C166%C2%AES-V2.htm>

Property	XC2734	ST10F272M
Max CPU clock frequency / Instructions per second	80 MHz / 80 MIPS	40 MHz / 20 MIPS
On-chip flash program memory	320 kB	256 kB
RAM memory	32 kB	20 kB
CAN Nodes / CAN message objects	2 / 64	2 / 64
ADC channels	9	24

Table 1 – Basic MCU specifications

Table 1 shows some of the key specifications which differ between the two MCUs. When comparing the remaining specifications and features between the XC2734 and the ST10F272M a lot of similarities are shown. This is quite natural since they both belong to the C166 family. The most important new feature for Haldex Traction introduced in the XC2734 compared to the ST10F272M is the support for JTAG debugging. See chapter 5.5 for a description of the technology and the possibilities introduced with this feature. Looking further at the specifications it's evident that the execution speed of the CPU has been increased. Not only by increasing the max frequency from 40 MHz to 80 MHz but the number of instructions executed per second at the same frequency is doubled. This gives the XC2734 an average execution speed of 80 million instructions per second (MIPS) when running at 80 MHz which effectively quadruples the value from the ST10F272M which delivers 20 MIPS at 40 MHz frequency.

3.3.3 Choice of microcontroller

One very important factor during development work of a new product is the cost associated. To cut the development time, and thus reducing cost, it's really important to be able to reuse both existing knowledge and existing design solutions which are proved to work. The new generation of HLSC doesn't demand a lot of new features in the microcontroller compared to the one in the current generation. This and the fact that Haldex has been using microcontrollers from the C166-family for a long time make it quite logical to look for a replacement of an old model among its successors in the same family.

The development environment for the Infineon processor can use a new revision of the currently used compiler called Tasking Classic. This means that the developers already know how it works and should be able to focus on the product instead of learning a new tool. Since the two microcontrollers are code compatible many parts of the currently used software can be reused in the new generation just by recompiling them for the new target.

3.4 Software

3.4.1 Rubus

The version of the operating system Rubus used in current Haldex Traction products is 2.0. To be able to see how a current version of Rubus OS worked on the MCU from Infineon an evaluation version of Rubus OS 3.4 has been used. *Worth noting is that the version of Rubus currently considered by Haldex is 4.0 which introduces another set of tools*. Rubus OS 3.4 and the tools supplied support a component based development model but this has not been used during the evaluation since it's outside the scope of this thesis. The approach taken is based on an attempt to reuse work done on earlier products to a large extent.

A couple of features introduced in this version which will affect the development process are:

- *The Green kernel.* This part of the OS provides a method to handle external asynchronous events like hardware interrupts. The advantages of letting Rubus handle hardware interrupts are among other things that the same threading concept can be practiced across the entire application. Keeping interrupts within Rubus also enables measurement of execution times of interrupt service routines from the OS.

- *Rubus Visual Studio.* The old version of Rubus was configured using a system of configuration files. The new version uses a program called Rubus Visual Studio to handle configuration. This tool enables viewing of resources allocated to an application in a graphical and hierarchical view.

Creating a Rubus OS application

In this section the author's experience with the Rubus OS is used to suggest a set of stages and a suitable workflow for creating an application for the OS.

1. *Planning.* To be able to create a good application it's essential to start with a planning stage where software requirements are specified. When these requirements are done it's possible to start planning the Rubus configuration. The OS requires control over at least one hardware timer which is used as a basis for all time measurement. The resolution of this timer should be decided based upon the timing demands of the tasks to be done. The nature of the tasks should also be studied to decide if they should be run as red threads with hard real time constraints. It's also necessary to specify which schedules should be created and their timing properties.

2. *Create a project in Rubus Visual Studio.* When the planning stage is completed it's time to create a Rubus project. A basic project in Rubus Visual Studio consists of two parts. The biggest part is called "Application" and this is where the resources of the different kernels (Red, Green and Blue) are specified. All threads, schedules and their timing properties are specified in this part. The second part is called a "Node" and basically

describes the target which the application will run on. This separation of application and target makes it easier to adapt an application to a new target since all settings regarding the target are stored in one place. When the user is satisfied with the settings it's time to build the project. Building a project means that Rubus Visual studio will try to generate a set of source code files which reflects the settings in the project. These files are used in the application to configure Rubus.

3. *Rubus specific coding.* The target hardware needs to be setup to match the settings decided upon in the planning stage and used in Rubus Visual Studio. There are also some functions which will need to be implemented to get Rubus OS running. These functions can be found in the Rubus OS Reference Manual. They are related to time measurement/management and how to handle errors, for example a deadline overrun.

4. *Application coding.* Once the Rubus specific coding is completed the developer should have a working execution platform. This allows programming and testing of the functionality of the actual application.

3.4.2 Base layer

The overall layout of the base layer is explained in chapter 2.3.3. This layout means that the hardware dependant code is placed in different driver modules which are relatively small. Adapting these modules to the suggested microcontroller should be fairly simple since its hardware layout is very similar to the current microcontroller. As long as the new driver modules provides the same calling interface as the old ones, reuse of the old manager and device modules should be possible.

The third-party CAN implementation will have to be upgraded to a version which supports the new microcontroller. In the application there's a layer between the CAN implementation and the rest of the application. This means that even if the third party supplier has introduced changes in the calling interface, adaptation to this will be kept in one place.

Apart from these changes there are three different modules which are based on assembler code. These modules provide startup routines, error detection regarding the RAM memory and error handling regarding stack handling. The startup routine will have to be rewritten to work with the new microcontroller. At the same time a more detailed documentation should be written. The error detection for RAM memory can be reused to a large extent but should also be more thoroughly documented. Finally when it comes to the module which handles stack errors, the author suggests that it's rewritten in C-code to make it easier to maintain.

3.5 Conclusions and discussion

The main conclusion drawn after the study of the current product and the XC2734 is that the Infineon processor is a good option for a new generation of the HLSC. The main reason is that the structure is very similar to the currently used ST10. This both simplifies the porting process and ensures that it's possible to reuse the currently used build tools. Worth noting is that no aspects of the electrical characteristics of the new microcontroller are weighed into this conclusion.

To continue using the Rubus OS with the new microcontroller it's necessary to switch to a newer version of the OS since the manufacturer doesn't support the microcontroller in earlier versions. This should be a minor risk since although some new concepts have been introduced it's possible to reuse the current implementation design.

Regarding porting work of the base software a rough estimate is that software development process to port the current application to the new microcontroller would take at most one man-year. This estimate is based on the work performed to create a Rubus application and the study of the current product. The author does not have extensive experience of projects in this size or detailed knowledge about the developing process used. Problems might appear in the control software which has been outside the scope of this study and the author's advice is to plan the project in cooperation with someone who has good knowledge about the higher levels of the application.

4 Program loading

4.1 Overview

The currently used system for program loading at Haldex Traction has been studied together with the documentation of the XC2734 microcontroller to determine which possibilities are available for this task with the proposed new microcontroller. Finally a demonstrator which shows a program loading concept has been built to verify that the theoretical results work in practice as well.

4.2 Available methods

To load a program into a microcontroller can be somewhat of a tricky process. One commonly used method is to put the processor in a special bootstrap mode. This mode is entered by setting the levels of a set of the processor's pins to a specific combination. In this mode the microcontroller can load code into its RAM-memory via a serial interface and execute it once the loading process is completed. This method is called *bootstrap loading* or *boot loading* and the code transferred to RAM is usually called bootstrap loader or just boot loader. The boot loader is commonly used to provide a platform from which a complete application can be written to a persistent storage, such as a flash memory.

Boot loading usually works fine preassembly but if the microcontroller is placed in a sealed housing it might not be possible to physically access the needed pins after assembly. In these cases the only viable solution might be to load an application before the final assembly of the unit. If there's a need to be able to load a new program after assembly, support for this must be built into the actual application. This is the method currently used in the Haldex LSC.

Another method is to use the JTAG interface, described in chapter 5, to load an application. This solution presents the same difficulty as the method above since it also requires physical access to pins of the microcontroller.

4.3 Current solution

The current application startup procedure is separated into two different phases. The first phase which occurs after power-on is called boot loading and is handled by the *Primary Boot Loader* (PBL). The PBL is independent of the actual coupling control application and has got two main responsibilities; to perform hardware initialization and also to provide a possibility to load a new application over the CAN interface. The PBL is considered to be a fixed piece of software which should not be the subject of change. It often resides in a flash block different from the application to ensure that it's possible to erase the application separately from the PBL. Since all units in a car must follow the same rules for program loading it's common that a PBL is supplied to subcontractors by car manufacturers. This both ensures that all units follow the same rules and also cuts development costs since the different subcontractors won't have to develop their own PBL.

This solution does not allow changing of the PBL and practically means that once an ECU has been loaded with an application and assembled that ECU will be branded towards a specific customer. For current generations of the HLSC, initial program loading is done by a Haldex subcontractor who supplies the ECUs. To decrease the number of different product variations from this subcontractor, and thus reduce the cost, the author has been investigating a new concept for program loading.

4.4 Suggested solution

To get the possibility to change the entire contents of the non-volatile flash memory there must be an optional storage location available from where it's possible to execute code. The best option is to use part of the internal RAM memory for this storage. The biggest drawback with this is that the contents of this memory will disappear if the power to the ECU is removed. The code which runs from RAM will have to be completely self supporting. This means that the RAM application must contain both communication drivers and provide a method to write data to the persistent flash storage.

Loading of a RAM application can be done by storing it together with a normal application in the flash memory and letting the application copy it to RAM and activate it. Another option is to let the flash application contain functionality to download an application over CAN and save it in RAM memory. Both of these options are viable solutions but the author favors the second one. The reasons for this is that it's possible update the RAM application without having to first re-flash an ECU and at the same time the RAM application won't occupy flash storage space.

Combining the ideas from the two previous paragraphs the author has implemented a demonstrator that includes both a PC-software running on Windows and target applications which allows changing the contents of the entire flash memory of an ECU. Two different applications have been designed for the target. One is a flash application which contains CAN drivers and provides the possibility to store an application in RAM memory and execute it. The second application is designed to be run from RAM and is completely self supporting. It also contains the drivers necessary to write information to flash memory.

The process of loading an application is divided into three stages.

1. *Load the flash application.* Two different scenarios are possible.
    a. The ECU is loaded with the flash application by the ECU supplier before assembly.
    b. A branded ECU is loaded with the application through the loading method supplied by the already installed PBL.
2. *Load RAM application.* In this stage the flash application downloads a RAM application by communication with the PC application over the CAN bus. Since this communication isn't compatible with other units in a car the ECU can't be

connected to the normal CAN bus. When the entire application has been downloaded execution is transferred to RAM.

3. *Load new application.* The old contents of the flash memory are erased and a new application is downloaded over CAN and written to flash. This new application can be for a different customer if the ECU were branded from the start. Once finished, execution is transferred to the new application.

The biggest risk with this process is coupled to the third stage of the process. If the ECU would experience a sudden power loss and restart there's a big risk that the flash memory won't contain a valid application and the contents of the RAM memory has disappeared due to its volatile nature. If this situation happens the ECU will most likely need to be restored by means of bootstrap loading.

To counter the risks the third step can be split into different parts

3a. Only delete enough space to allow a new PBL to be loaded and make sure that the flash application or the interrupt vector table are not deleted. The interrupt vector table is an address table which among other things specifies where the processor will start execution after a restart.

3b. Write the PBL and verify that it has been correctly written.

3c. Delete the interrupt vector table and rewrite it pointing the restart address to the PBL instead of the flash application.

3d. Continue erasing flash contents and write the rest of the new application.

This method would limit the time the ECU is vulnerable to a power loss. If a power loss would be experienced prior to *3c* it's possible to restart the process from *step 1*. If the power loss would occur after *3c* the PBL has been loaded and it's possible to load the ECU using the method specified by the PBL. The time the ECU will be vulnerable is decreased to the short amount of time needed to erase and write a very small amount of code. In a production environment this risk could be decreased by using some kind of failsafe power supply.

4.5 Conclusions and discussion

The work with the demonstrator application shows that the suggested solution is possible from a technical point of view. Further work in this area should be focused on creating a suitable protocol for program downloading over CAN and improving the flash memory routines.

Although there are risks involved in the suggested process these will have to be weighed against the advantage of only having to order a single product from the ECU subcontractor and being able to "rebrand" ECUs.

5 JTAG-Debugging

5.1 Introduction

During the 1980's surface-mount technology became a widely used method for constructing electronic circuits. This method allowed components to shrink in size but also made testing of mounted components much harder. Traditionally testing had been done by the use of a bed-of-nails tester where the circuit board is pressed against small measurement probes which make contact with test points on the board. This was not a feasible solution when surface-mounted components started to be placed on both sides of the circuit boards and the size of components continued to shrink. To solve this problem and allow continued testing of components the Joint Test Access Group (JTAG) was formed by a group of electronics companies. The group devised a specification for boundary scan hardware testing which provides an integrated method for testing interconnects at the integrated circuit level. This allowed test engineers to test for defects such as unconnected pins or a failing device and the specification was turned into an international standard, the IEEE 1149.1.[3]

5.2 Debug

While the IEEE 1149.1 standard was created to allow testing without using a lot of probes the JTAG pins of many microcontrollers have a secondary function, usually referred to as JTAG debug mode. In this mode the JTAG-interface is used as a high speed serial bus to access the on-chip debug engine of the CPU. The debug engine is often used to set breakpoints in the program and provide access to processor registers and peripheral units. Additionally it's common during development to load program code into non-volatile memory with help of the JTAG debug mode.[4]

5.3 Components

A basic setup for microcontroller debugging through JTAG consists of the following components:

- *Target.* The microcontroller to debug.
- *Host computer.* A conventional computer for example a standard PC.
- *Communication hardware.* A hardware which converts the raw signals from the JTAG interface on the target to a format which is suitable for the host computer to read. A common term for this unit is *wiggler* which originates from the name of a commercial JTAG tool.

[3] Corelis Corporation, *Boundary-Scan Tutorial,* retrieved April 29, 2009, <http://www.corelis.com/products/Boundary-Scan_Tutorial.htm>

[4] Direct Insight, *Boundary Scan / JTAG,* retrieved April 29, 2009, <http://www.directinsight.co.uk/function/boundary-scan-jtag.html>

- *Debugger software.* A software running on the host computer which is used to access the on-chip debug capabilities of the target. Physical connection between the host and the target is made through the communication hardware.

5.4 Alternatives

Debugging through the JTAG interface is not the only available option. Device Access Port (DAP) is another interface which is specific to the semiconductor manufacturer Infineon and it might be an option in the future. Today there's not enough information readily available to evaluate this option.

A second alternative is to use a monitor, a program which exists side by side with the regular application in the code memory. The monitor has the possibility to control the On Chip Debug Support (OCDS) system of the microcontroller. Control of the monitor program could be done through any available communication means, i.e. CAN. The biggest drawback with this is that monitor must be loaded into program memory thus reducing the amount of memory available to the application.

5.5 Evaluation of different JTAG debugging systems

5.5.1 Introduction / Methodology

An evaluation has been done of four different JTAG debugging systems. Each one of these systems consists of communication hardware and debugger software. The features of the communication hardware have been briefly studied to determine that the basic demands are fulfilled but the largest emphasis has been put on the debugger software. During the work with this thesis the author has used the different software for common debugging purposes. The software has been tested both at a relative high level together with Rubus OS and really close to the hardware during the work with porting of TinyTimber described in section 6. The practical tests for all systems but the one from Lauterbach have been done using an on-board USB-wiggler on an Infineon XC2000 family starter kit board.

5.5.2 Lauterbach In-Circuit Debugger

Lauterbach is the largest manufacturer of emulators and debuggers worldwide and their debug concept is a bit different than the other products in this evaluation. The debug software is called Trace32 and it's mainly run on a separate debug interface coupled to the host computer through e.g. Ethernet or USB. The host system runs a small part of the software and basically acts as thin client. This concept allows a system which can be run on many different kinds of hosts but from a user's point of view this behavior is transparent.[5]

[5] Lauterbach GmbH, *TRACE32 Microprocessor Development Tools,* retrieved May 1, 2009, <http://www.lauterbach.com/main.html>

Ease of use

Trace32 is a complex application with possibility to customize almost everything. To setup a debug session the first time a user needs to read a fair bit of documentation to set the correct options. Once all options are set correct it's possible to save them to a batch-file which utilizes a script language called Practice. Using this language it's possible to do everything that's possible from within Trace32 but also control program flow and handle files.

Figure 6 - Trace32 Command Line

One interesting and helpful feature of Trace32 is the Command Line (shown in Figure 6) where all available commands can be typed in and settings changed. Below the Command Line are context aware buttons, called *soft keys*, which helps the user see available options and they can also be clicked instead of typing a command. Tab-completion when writing commands is available to facilitate input.

Functionality

Trace32 has got a lot of functions which are beyond the scope of this comparison. One feature which separates it from the other products is the support for the real-time operating system used currently by Haldex, named Rubus. This allows the user to see which thread is running and watch the values of operating system dependent variables. The figure below shows some of the information which is available. Most values are zero since the screenshot was taken in a simulated environment without a connection to the actual target hardware.

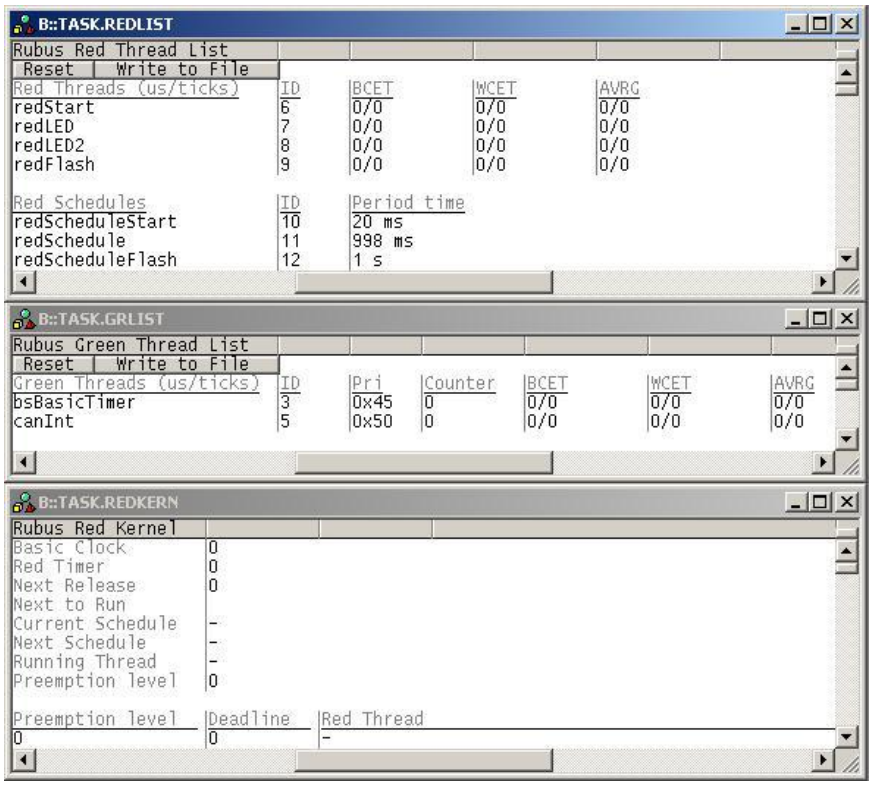
B::TASK.REDLIST
Rubus Red Thread List
Reset | Write to File
Red Threads (us/ticks) ID BCET WCET AVRG
redStart 6 0/0 0/0 0/0
redLED 7 0/0 0/0 0/0
redLED2 8 0/0 0/0 0/0
redFlash 9 0/0 0/0 0/0
Red Schedules ID Period time
redScheduleStart 10 20 ms
redSchedule 11 998 ms
redScheduleFlash 12 1 s
B::TASK.GRLIST
Rubus Green Thread List
Reset | Write to File
Green Threads (us/ticks) ID Pri Counter BCET WCET AVRG
bsBasicTimer 3 0x45 0 0/0 0/0 0/0
canInt 5 0x50 0 0/0 0/0 0/0
B::TASK.REDKERN
Rubus Red Kernel
Basic Clock 0
Red Timer 0
Next Release 0
Next to Run
Current Schedule -
Next Schedule -
Running Thread -
Preemption level 0
Preemption level Deadline Red Thread
0 0 -

Figure 7 - Trace32 support for Rubus OS

Overall impression

The knowledge threshold before a user is able to use this application effectively is fairly high. Once familiar with available commands, common tasks can by executed fast by using the command line. It has got support for non-ambiguous abbreviations, e.g. the command "DATA.LIST" can be abbreviated to "D.L". The command line is also aware of the currently loaded debug information which means function names can be specified directly in commands with the help of tab-completion.

One drawback is that it's easy to end up with a lot of different windows on the screen after a while. This sometimes makes it hard to follow the program counter between different functions and windows. The look and feel of the user interface and overall layout is a bit old fashioned. The products had no problems with stability during the evaluation period.

5.5.3 Tasking CrossView Pro 8.7R3

The company Altium is the maker of the compiler used at Haldex and their product portfolio for embedded software, called Tasking, also includes CrossView which is their debugging software.

Ease of use

CrossView is pretty straightforward to use, most common functions are accessible through icons which are pretty self explanatory. Similar to Trace32 the application has got a command line

where it's possible to enter commands but the user don't get any help at all to see which commands are available which means this feature will most likely not be used. When looking at CPU registers it's possible to define a set of registers which should be visible in a window. This helps reduce the overhead imposed by having a large amount of windows.

Functionality

The software contains all standard functions a user can expect from a debugging application. Apart from this it's possible for the user to write own macros in a C-like language.

Overall impression

Most basic functions are really intuitive and easy to use. The more advanced functionality is harder to find and feels clumsier, e.g. looking at different parts of the memory which is unnecessary complicated to setup. During the evaluation no stability issues were encountered with this software.

5.5.4 Hitex HiTOP 5

HiTOP is a universal user interface for all of Hitex's test and analysis tools. The product which enables JTAG debugging for the considered microcontroller is called TantinoXC.

Ease of use

Usage of basic functions is simple and many programmers will recognize and understand the graphical user interface intuitively. Setting a breakpoint with conditions is not trivial and the help-function in the author's version doesn't correlate entirely to what's visible in the program. Viewing of CPU and peripheral registers is not easy since the information is spread amongst several different pages which the user will have to browse through to find the relevant information. See Figure 8 which shows twelve pages all named SCU. It's possible to use a program called "SFR Editor" to change the appearance of the registers and values.

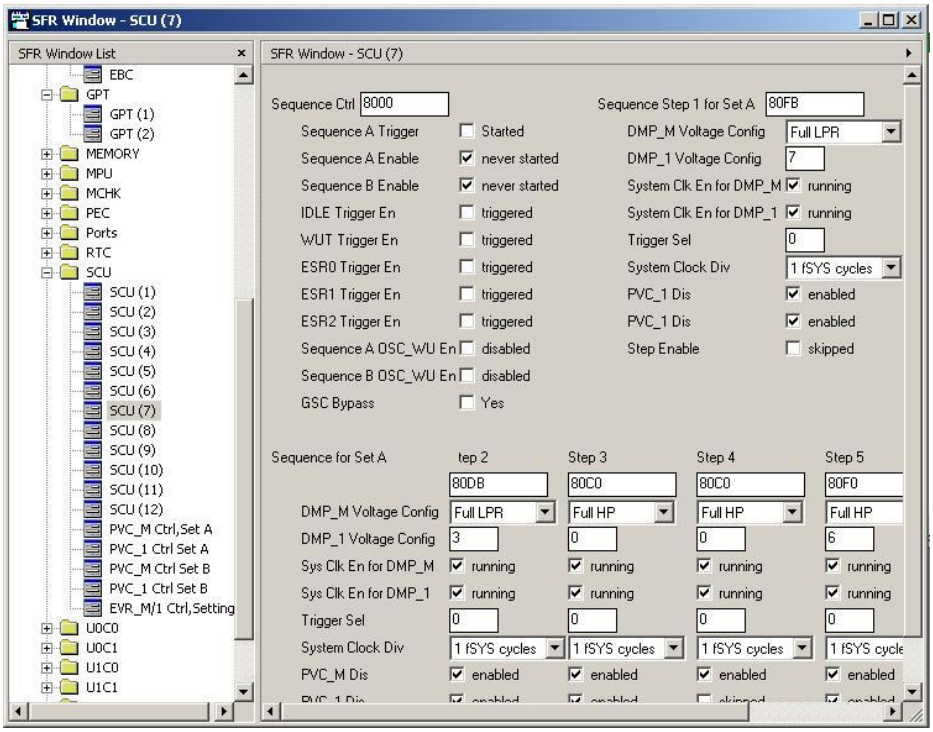

Figure 8 - Special Function Registers in HiTOP

Functionality

Apart from offering debugging HiTOP also features an integrated development environment (IDE) which means it's possible to write code, build it and debug the code within the same application.

Overall impression

The program was easy to setup and get started working with and the user interface feels modern. Simple breakpoints are created by clicking in the left margin of a code line, a concept which is familiar to most users of debugging software. One big error in this version of the software is that the value shown for a timer reload register didn't correspond to the actual value in the register. This was proven by switching to another application where everything looked correct. Another drawback is that the application crashed a few times while loading debug information for a program.

5.5.5 Universal Debug Engine (UDE) 2.04.11

UDE is created by a company named pls Programmierbare Logik & Systeme GmbH. Regarding communication with the hardware device two different suitable products are available. They are called UAD2 and UAD2+ where the latter has got more features.

Ease of use

Basic functionality is easy accessible and works as expected. Most aspects of the layout of the application's windows are customizable. This means that the user is able to set a certain layout behavior but the multitude of options might be a bit confusing.

Functionality

A button which allows jump to the current location of the program counter helps when having a few different windows open. There's also an option available which switches between showing the whole program in a single window or split it into different windows for the different modules of the program.

Overall impression

Viewing of registers is one basic task which holds plenty of room for improvements in this application. The default settings provided a confusing view with a lot of text in too small text fields. It's possible to change the layout a bit so all of the text can be shown but it still feels clumsy. When changing size of the columns it was possible to cause a situation where a column was set to the smallest size which meant that the content could not be read and demanded a reset of the layout to be visible again. Figure 9 shows two different view layouts of special function registers in UDE. The possibility to let a single window float outside the main window can be useful when working with multiple displays. Overall the application is pretty easy to work with but it did crash a few times while loading or closing a project.

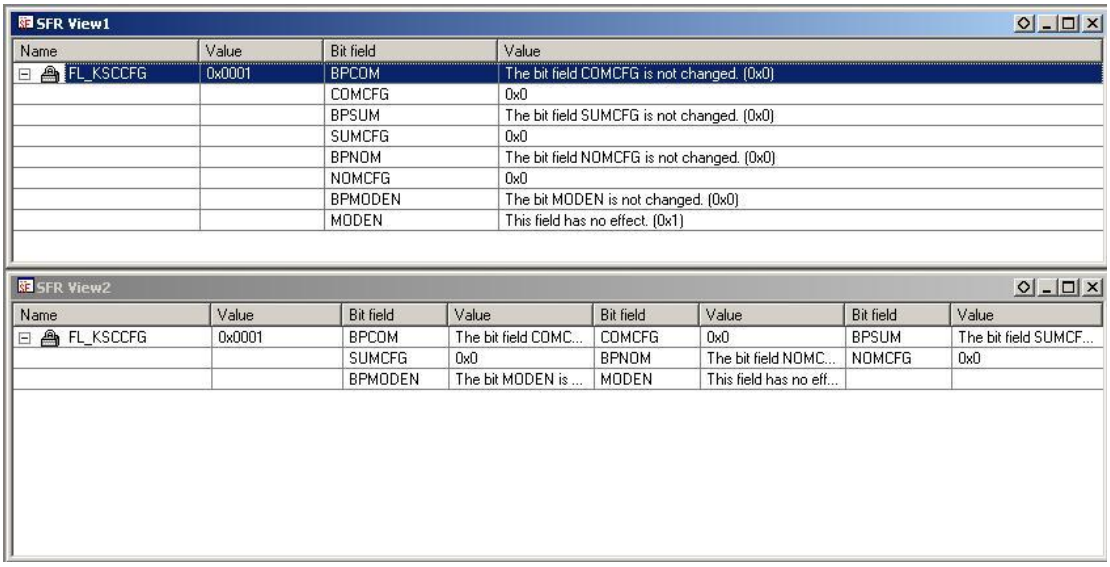

Figure 9 - Two different register view layouts in UDE

5.6 Conclusions and discussion

JTAG debugging is a very useful tool for an embedded developer. Without debugging possibilities development can be like navigating a china shop in complete darkness. Something is bound to break and when it does it might be very hard to see what has happened.

Each one of the evaluated applications and their associated hardware provides functionality which will facilitate the development of new products at Haldex. When comparing the four applications side by side the applications HiTOP from Hitex and UDE from pls don't feel as mature as the other two. Crashes were experienced with both of these applications which are a big deterrent factor. Regarding UDE the behavior when viewing registers was annoying and it feels like more focus should have been put on providing a good layout from the start instead of having the possibility to change so many aspects it. Viewing of registers in HiTOP was disappointing as well. This might seem like a trivial aspect but is important, especially when writing drivers or other code which works in close contact with the hardware. CrossView Pro provided the wanted functionality with no big drawbacks but at the same time it didn't have any functionality which set it aside from the other products. The product the author recommends is the one from Lauterbach. The two biggest reasons for this are:

- Engineers at Haldex are already familiar with the software since it's been used during development of earlier generations of the HLSC. Lauterbach is also the biggest manufacturer of debuggers and emulators which means that if need appears to expand the engineering staff there's a greater chance to find a person who has got previous experience with the tool.

- The support for the real time operating system Rubus is useful although it is possible to retrieve the same information in the other applications. The difference is that the user will have to either build some kind of extension to the application or look up the information manually and both options are time consuming compared to using an already available solution.

6 TinyTimber

6.1 Introduction

TinyTimber is a small real-time kernel developed at Luleå University of Technology (LTU). It implements a subset of a design paradigm for embedded systems called Timber.[6] Timber is based around a notion of reactive objects which remain in a passive state until they receive stimuli in form of events from the execution environment.[7] The concept of reactive objects and Timber are active research subjects at LTU and both Timber and TinyTimber are used by students in courses at the university.

While Timber specifies a new programming language, TinyTimber provides an interface to utilize the same reactive programming paradigms in standard C-code. For more information about Timber and TinyTimber see the licentiate thesis by Eriksson[6] and the Timber language homepage[7] which provides an easy to understand introduction and lists a set of relevant publications.

The TinyTimber kernel is available to a limited set of target platforms. The author has adapted the kernel to work with the XC2000-series of microcontrollers considered by Haldex for the next product generation. The resulting kernel can be used for practical tests with the Infineon microcontroller and thus provide a basis for a more direct comparison between the currently used Rubus OS and TinyTimber. This chapter will provide a description of the implementation and information which will facilitate similar work for other target platforms.

6.2 Software structure

The TinyTimber software is divided into two different distinct parts. One is the part of the kernel which is completely target independent. It provides the TinyTimber calling interface and performs scheduling and handling of threads and messages. The other part is called an environment and this is where all target specific code is stored.

TinyTimber uses the GNU Make toolset[8] to control the building process and it's possible to build an application for a different target platform by changing a single environmental variable. The makefiles supplied are created for a Linux/Unix environment and requires some minor changes to work as expected in a Microsoft Windows environment.

6.3 Time management

Keeping track of time is one of the most important tasks in a real-time operating system. Operations performed by the system can have restrictions both on the time when they may start and when they must be completed.

---

[6] J Eriksson, *Embedded Real-Time Software using TinyTimber – Reactive Objects in C, ISSN 1402-1757,* Luleå University of Technology 2007.
[7] Timber, *A gentle introduction*, retrieved May 15, 2009, <http://www.timber-lang.org/index_gentle.html>
[8] Free Software Foundation, *GNU Make,* 2006, retrieved May 15, 2009, <http://www.gnu.org/software/make/>

The easiest approach to keep track of time is to use a tick-based system timer. This means that a periodic timer is setup to notify the kernel each time it reaches its target value. When this happens the system timer is increased one tick. Scheduling of operations is made with tick-precision. The biggest drawback with this method is that the user will have to balance the need for precision against the overhead created by calling the kernel each time the timer overflows.

A better option is to setup a timer to expire when the next operation should begin and notify the kernel when this happens. This might sound like the most obvious solution but the implementation is more complicated than the tick-based approach. Correctly implemented this method provides much better precision with lower overhead. This is the approach taken when implementing an environment for the XC2000, see section 6.5.

6.4 Creating a new environment

6.4.1 Pre-study

The first and most important step when creating a new environment is to perform a study of the different components in the system. The main areas which are important to understand are the target platform, compiler, assembler and the TinyTimber system. Following is a more in-depth explanation of what information to look for and why for each one of these components.

Target platform

It's obvious that a developer need to study the target platform when planning an application. A fairly good knowledge about all parts of the target is necessary but some parts require a really deep understanding. Timer and interrupt handling are two very important areas when writing an environment for TinyTimber. Where is the return address stored when an interrupt occur? Do the general purpose registers get saved somewhere? These are a couple of questions which the developer will need to answer. Apart from the examples above, stack handling functionality and different addressing modes are two other important subjects.

Compiler and assembler

Since some parts of the environment most likely will have to be implemented in assembler it's necessary to understand how the compiler and the assembler interact. Knowing how the compiler handles interrupt service routines (ISRs) should be another area of study. For example the Tasking Classic compiler for the C166 processor family inserts a special interrupt stack frame in ISRs, but it's also possible to use a customized stack frame.

TinyTimber

Having a complete understanding of the inner workings of the kernel is a key aspect to be able to provide an adequate environment for it to operate in. If possible, running a TinyTimber application on an already supported system with debugging possibilities should provide an excellent learning platform. If using another target isn't a viable option, studying an environment for another platform and "running it" on paper is a good idea.

6.4.2 Implementation

Supplied with the TinyTimber source code is a skeletal environment which can be used as a template for a new implementation. It's possible to compile the skeleton environment but since it's a phony implementation it does not run. Building this phony environment is a good exercise since it ensures that the build tool-chain is reasonable configured. Any needed changes should be done in the build settings specific for the new environment.

Some important decisions will have to be made at the very beginning of the implementation process. Where to store and how to handle the kernel stack is one of these issues. Depending on the target architecture the available options will vary. The address range where the stack can be stored might be limited. Another issue to be aware of is that the size of the stack is dependent on the number of execution threads the kernel should support. This means that even if an environment works excellent with the stack placed in a limited memory area, increasing the number of threads might render the same implementation incorrect. One idea used in some of the already implemented environments is to create a global available array with a size dependent of the number of threads and use this array as stack. The biggest advantage of this approach is that the problem of placing the stack is moved to the build tool-chain. Although a convenient method, care have to be taken to ensure that the array isn't placed outside the range of the stack pointer of the microcontroller.

Time measurement is another aspect which requires some thought. The developer is free to provide a completely own solution to this problem. One demand is that the implementation calls the two kernel functions `tt_expired` and `tt_schedule` when the baseline (start time) of a message is met. Although it might be possible to create an implementation which only uses a single hardware timer the author recommends an approach where a combination of a timer and some kind of hardware compare functionality is used. Hardware compare means that the value of a timer is monitored by the hardware and if it matches a set value an interrupt is raised. Using this approach the timer can be used as a periodic time base and the compare functionality can be used to mark baselines of messages in the period.

Once these aspects have been addressed it's possible to provide an implementation for most of the functions in the skeleton environment. Two functions which require a lot of care are `ENV_CONTEXT_INIT` and `ENV_CONTEXT_DISPATCH`. The first one is used to perform initial setup of the stack for each individual thread. The second function is used to switch between different contexts. A context switch means that the status of the currently running thread is saved and the status of a new thread is loaded and execution is continued in this new thread. This is probably the most difficult aspect of the implementation. Knowing which information needs to be saved when leaving a thread to be able to return to it is of utmost importance. Context switches will need to be performed both while running in a normal mode and from an interrupt service routine (ISR). Thus the information stored when leaving a thread in normal mode must be enough to be able to restore it from an ISR and vice versa. These operations will most likely need to be expressed using inline assembler.

6.5 Details of the XC2000 environment

Each microcontroller requires its own solution when it comes to hardware dependant functions such as timers and handling of interrupts. This section will describe details of the implemented environment and the microcontroller hardware.

6.5.1 Time-management

Some kind of timer or clock is a necessity to be able to measure time. In the XC2000 series of microcontrollers a type of unit called Capture/Compare Unit (CCU) is available. It provides the possibility to continuously compare a chosen value with the current value of a timer and trigger an interrupt when they match. This feature is used in the environment to create a stable time management system. A timer is setup to have a period of 10ms, i.e. it generates an interrupt every 10^{th}ms. When a timer interrupt occurs the Interrupt Service Routine (ISR) checks if any action should be taken during the new period. If this is the case the value of the CCU is setup to match the time of the action. When the CCU interrupt occur scheduling functionality of the kernel is called.

6.5.2 Interrupts and context switches

Interrupt handling is the process which provides a response to an asynchronous event in a microcontroller. It interrupts normal execution to execute a specially designed function called interrupt service routine (ISR). One example could be a timer which reaches its target value; the timer's ISR provides a reaction to this event which could be as simple as toggling a LED. To be able to continue execution normally when the ISR is finished some information is stored automatically before the ISR is entered and restored upon exit. The process is called context switching and the information stored is called interrupt context frame and it's normally saved on the stack.

An application which is to complete a set of tasks with some time-constraints might be forced to use the concept of context switching to fulfill its task. One example is an application which must execute two tasks within 100ms. The first task takes 90ms to execute and the second one takes 10ms but must be executed within 40 to 60ms from the start. The only way for the application to fulfill these requirements is to start executing the first task and then switching to the other one and finally switch back to the first again.

Context switches in the TinyTimber environment can be triggered from two different places. Either from the ISR of the CCU or as a function call from the kernel. In the XC2000 environment a context switch consists of three different steps. First information about the old context is saved. The information saved is for example contents of the general purpose registers of the processor. Secondly the new context to activate is selected and finally the new context is loaded by reading back all information stored for the context. It's possible for a context which has been saved in an ISR to be restored from a normal function. This imposes a demand that the same information is stored independent of where the context save is initiated from. As described in the first paragraph an interrupt context frame is saved on the stack before entering an ISR. This means that the starting point in an ISR won't be the same as from a normal function. Apart from

the information automatically saved by the microcontroller the Tasking Classic compiler saves additional information when entering an ISR. To circumvent this, the implemented environment creates a fake interrupt context frame when called from a normal context. The fake interrupt context frame contains the same information as is stored automatically by the microcontroller and the compiler in an ISR. When restoring context from an ISR nothing special needs to be done since all information is available. Restoring context from a normal function requires handling of the extra information added in the first step.

6.5.3 Suggestions for future work

The most complicated part of the environment is the context switching. This is also the part of the implementation which holds room for the biggest improvements. In the XC2000 series of microcontrollers it's possible to select a memory address where the general purpose registers will be stored. When choosing a new address the old content of the registers are saved and new values are loaded. This could be used to provide a more elegant context switch instead of saving the registers one by one on the stack.

Another area where work could be done is the conversion between different addressing modes. A complete address in the microcontroller is 24 bits long but each register is only 16 bits long. To obtain a complete 24 bit address the processor uses different modes and the support for conversion between them should be extended. See the XC2000 User's Manual for more information about how the different modes work.

6.6 Discussion

The methodology introduced in this chapter is the author's own idea about a good workflow. The exact same method was not used during the actual implementation of the environment but is based on lessons learnt from that implementation work. As such the hope is that it will help another developer avoid some of the pitfalls involved.

References

Corelis Corporation, *Boundary-Scan Tutorial,* retrieved April 29, 2009, <http://www.corelis.com/products/Boundary-Scan_Tutorial.htm>

Direct Insight, *Boundary Scan / JTAG,* retrieved April 29, 2009, <http://www.directinsight.co.uk/function/boundary-scan-jtag.html>

EmbeddedDeveloper.com, *Infineon C166®S V2,* retrieved April 3, 2009, <http://www.embeddeddeveloper.com/cores_variant/66/Infineon-C166%C2%AES-V2.htm>

Eriksson J, *Embedded Real-Time Software using TinyTimber – Reactive Objects in C, ISSN 1402-1757,* Luleå University of Technology 2007.

Free Software Foundation, *GNU Make,* 2006, retrieved May 15, 2009, <http://www.gnu.org/software/make/>

Lauterbach GmbH, *TRACE32 Microprocessor Development Tools,* retrieved May 1, 2009, <http://www.lauterbach.com/main.html>

STMicroelectronics, *Datasheet ST10F272M*, 2008, retrieved May 16, 2009, <http://www.st.com/stonline/products/literature/ds/12968.pdf>

Timber, *A gentle introduction*, retrieved May 15, 2009, <http://www.timber-lang.org/index_gentle.html>